(D R A F T – v0.4)

Proposal for a Strongly-Typed database API for CMS

S. Kosyakov, J. Kowalkowski, L. Lueking, P. Maksimović, M. Paterno, L. Sexton-Kennedy, and R. Wilkinson

April 8, 2005

Abstract

We present a proposal for an implementation of the DB API in ORCA based on Strong Typing (ST) function calls. In the ST API, each query type (e.g. a request for a set of HCAL pedestals) is fixed prior to the beginning of the job: for the same query type the amount of information requested is the same from call to call, and the returned DB object always have the same structure, which is furthermore written by a code-generator and thus compiled into the executable. This approach can be contrasted to Attribute List (AL) in which the client's requests are dynamic in nature and can vary greatly from call to call for the same query type.

1 Introduction

The DB API is intended for all ORCA and OSCAR executables, from the HLT farm (filtering, monitoring, and possibly corrections to calibrations) to the offline event reconstruction (‘production’), user analysis, and generating of ‘realistic’ Monte Carlo jobs (using OSCAR) which also use the detector conditions (calibrations, trigger tables, etc.) to make the simulated detector output look like real data.[1]

This note thus does address other types of DB access through XDAQ and other stand-alone tools. Moreover, it is largely concerned with reading of the DB constants because that constitutes the majority of the DB access activity in ORCA executables.

The following pages outline a concrete proposal for a strongly typed interface based on a code-generation.

1.1 Definitions

It is useful to define some commonly used terms. A *conditions set* is a complete set of calibrations for all DAQ channels (calorimeter cells, tracker silicon strips, pixels. . .). Within a conditions set, the DAQ channels are grouped into *records*. This grouping is very detector-specific and usually represents an physically indivisible piece of detector; for example, the pedestal for each HCAL cell may be stored in a record, however all pedestals in tracker’s silicon wafer could be stored together.[2]

Within one conditions set, each record is identified with a *record key*, which is unique. Each record key has one-to-one mapping to each detector element, and therefore represent a natural connection between the database, the DAQ, the geometry, and a specific detector numbering scheme. The *uniqueID* may be a natural choice for the record key.

We note that the alignment constants can also be stored in condition sets since the unit of information is still a detector element, except that each record stores the alignment information (translations, rotations, and deformations) instead of pedestals.

A *query type* is a request for a certain kind of database content, for instance, a request for a calibration set of HCAL pedestals, or a request for a certain trigger table. As used in this document, a *query* is a specific request of a given query type, *e.g.*, a request for HCAL pedestals for run=1732579.

Almost all ‘DB read’ activity can be divided into two broad use-cases:

- **Calibrations**: the client needs a full set of calibrations for all detector channels. One conditions set must be identified uniquely in the DB, and thus will be requested by a *conditions ID* (CID). The majority of the DB content is obtained in this way.
- **Free-form SQL** for special purpose queries which cannot be shoehorned into the ‘Calibration’-type access pattern.

[1]Indeed, at CDF the realistic MC production is by far the most demanding client of the DB content.

[2]This approach is used at CDF, since, for 700,000 channels, the access is about two orders of magnitude faster than when each channel is stored individually.

2 Overall architecture

There is some confusion regarding how the various parts of the DB API interact, and whether the client code should have any contact with the exposed interface of an underlying transport mechanism (e.g. Frontier, ODBC client...). We propose a division of DB API into three distinct areas:

- High-level API
- Code-generated code for each type of query
- Underlying transport mechanism ('back-end')

Each one is described in the subsections below.

2.1 High-level API

The High-level API groups common functionality into a centralized location (set of objects). This group of objects provides services that are needed for all queries, for example

- which database(s) (Frontier servers) to contact; for ODBC/Oracle/MySQL this could include the session management as well;
- which transport mechanism is used;
- what are the validity ranges of the used calibration sets (aka IOVs);
- (optional) callback to the client code once the new calibrations need to be – or have already been – loaded.

All of these features could be controlled by query type.

While it is obviously possible to embed all this functionality in each type of query (DB object, table), separating this kind of high-level code provides for a cleaner interface and reduces the physical coupling between the components, which can then be developed in parallel.

There are clear proposals how to manage IOVs at CMS, and also solutions from elsewhere (BaBar, POOL RAL), and thus this will not be discussed further here. For the discussion of code-generation, we assume that all necessary services of this kind are available from a centralized location.

2.2 Different database interfaces

The code-gen uses a base class, `DBReader`[3], which describes the interface used for reading. A `DBWriter` class will have a similar interface and could be used for writing. (They could be bundled together, although there may be some advantage of keeping them separate since `DBReader` and `DBWriter` may talk to different physical databases/servers.)

`DBReader` provides the following abstract interface:

[3]All class names are negotiable.

```
   int setup(DBTableDetails & details);
   int nRecords();     // the total number of channels, cells, chips...
   //
   char   getByte();
   int    getInt();
   long   getLong();   // actally long long
   float  getFloat();
   double getDouble();
   //
   char*  getByteArray();
   int*   getIntArray();
   long*  getLongArray();
   float* getFloatArray();
   double* getDoubleArray();
```

`DBTableDetails` contains the details that various DBReaders need to know, like the pieces of SQL, location of Frontier servers, Oracle database name, etc. (some can also be provided by the High-Level API service).

Frontier client already looks like a DBReader. Oracle/ODBC client has `operator>>()` overloaded for all data types, so it would be trivial to turn into `getInt()`, `getDouble()`, `getByteArray()`, etc. This approach will be implemented at CDF. Something similar is also needed by POOL RAL[4], and might be a good starting point for CMS.

Here is an example of using a generic `DBReader` in a code-generated client code for a (hypothetical) table `HcalCellPed` containing pedestals and noise for every HCAL cell. For the sake of the argument, they are both double.

```
void HcalCellPed::fetch( DBContainer<HcalCellKey,HcalCellPed>
                         & cell_container )
{
   DBReader * reader = getReader();     // a FrontierReader, ODBCReader, etc.
   reader->getDBContent( getTableDetails() ); // query the DB
   while (! reader->atEnd()) {                // iterate over records
      HcalCellPed oneRec;                     // one cell (value, error)
      HcalCellKey key ( reader->getInt() );   // build the key: the cell
      oneRec._value = reader->getDouble();
      oneRec._error = reader->getDouble();
      cell_container.insert( std::make_pair(key,oneRec) );
   }
}
```

This approach essentially uses the same data structure as `CaloCalibration` object from CaloCalibrator package [1]. However, using a map and an intelligent key (`HcalCellKey`) saves us from needing one extra class to perform that mapping. This way, the user can simply supply a cell id (an integer) into the map and use it to find the appropriate cell.

[4]And it might already been implemented.

```
  typedef DBContainer<HcalCellKey,HcalCellPed> HcalCellContainer;
  ...
  HcalCellContainer cell_container;       // storage for DB output
  HcalCellPed::fetch( cell_container );   // fill it up (query DB)

  HcalCellKey findMe( 12345 );             // let's find this HCAL cell

  HcalCellContainer::const_iterator cell_it = cell_container.find( findMe );
  HcalCellPed & aCell = cell_it->second();
  double value = aCell.value();
  double error = aCell.error();
  ... etc.
```

Since `HcalCellKey` class is coded by hand, it may contain all sorts of other useful information, like relationship with geometry (*e.g.*, we could supply indices of eta and phi cells in HCAL instead of a global id), DAQ, `uniqueID`, etc.

Occasionally, the data will be highly compressed (bit-packed) in the database, and the record obtained in the way described above, is not completely ready to be used by the client code. The compressed data needs to be unpacked. For example, consider a table `TkStripPed` which contains pedestals and noise for every chip in the Tracker. They are stored in arrays of bit-packed bytes in the database.[5]

TkStripPed is code-generated as usual, but the source for code-generation (e.g. an XML file) contains a flag which adds the post-query configuration into the template. The user also needs to supply a class `TkStripPedConfigurator` which is instantiated at the beginning of the job and a `DBTableDetails` object for this table points to it. (There are other ways to achieve this, but the idea is to be able to call some user-supplied piece of code within the loop over record in the code-generated `fetch()`.)

```
void TkStripPed::fetch( DBContainer<TkChipKey,TkStripPed>
        & strip_container )
{
   DBReader * reader = getReader();     // a FrontierReader, ODBCReader, etc.
   reader->getDBContent( getTableDetails() ); // query the DB
   TkStripPedConfigurator * conf = getTableDetails()->getConfigurator();
   while (! reader->atEnd()) {            // iterate over records
      TkStripPed oneRec;                  // one chip worth of peds + noise
      TkChipKey key ( reader->getInt() );  //  build the key: chip number
      oneRec._chipPedestals = reader->byteArray(128);
      oneRec._chipNoise     = reader->byteArray(128);

      if (conf)                        // <--- if the configurator is defn'd
          conf->configure(oneRec);    // <--- call user's code to unpack arrays

      strip_container.insert( std::make_pair(key,oneRec) );
   }
```

[5]For example, this is how it is done for CDF.

```
}
```

2.3 API for each query type

There's a lot of similarity of the `fetch()` methods in the above examples. The code is different only regarding the names and types of data fields in each of the classes (`HcalCellPed` and `TkStripPed`), which are different because they represent different columns in the database tables. And while the polymorphism here does not work, the code can still be shared by code-generating it from the same template.

Within the Strongly Typed approach, all access to conditions sets (keyed off uniqueID) is code-generated. The code-generation could start from

- a client C++ code (like POOL RAL)
- tables in the DB (various free or commercial tools)
- or yet another format (e.g. XML)

Starting the code-generation with an XML file is the most flexible of the three. The XML file (like Frontier's XSD) would contain

- variable names and types
- pieces of SQL necessary to build the query
- could satisfy both OTL/ODBC and Frontier
- other stuff (for ROOT, etc.) could be added as well

The 'pieces of SQL statements' (the "where" clause, "order by", etc.) will be compiled into the executable. Thus the SQL will exist within the executable, however it will not appear within the C++ client code controlled by the user.

Note that `DBReader` classes will need to know about `DBTableDetails` objects, since most of the reader-specific information each reader obtains from the `DBTableDetails` directly. One example of such information is the interval of validity: the reader will ask the high-level API for the conditions id (CID) of the table that needs to be queried, and use it appropriately. (ODBC will insert it into a SQL string, while Frontier will get it as a URI parameter. . .).

3 Bibliography

References

[1] `http://www.hep.caltech.edu/ rickw/FrontierCondDB/`